

HPAT: High Performance Analytics with Scripting Ease-of-Use

Ehsan Totoni

Intel Labs, USA
ehsan.totoni@intel.com

Todd A. Anderson

Intel Labs, USA
todd.a.anderson@intel.com

Tatiana Shpeisman

Intel Labs, USA
tatiana.shpeisman@intel.com

Abstract

Big data analytics requires high programmer productivity and high performance simultaneously on large-scale clusters. However, current big data analytics frameworks (e.g. Apache Spark) have high runtime overheads since they are library-based. Given the characteristics of the data analytics domain, we introduce the High Performance Analytics Toolkit (HPAT), which is a big data analytics framework that performs static compilation of high-level scripting programs into high performance parallel code using novel domain-specific compilation techniques. HPAT provides scripting abstractions in the Julia language for analytics tasks, automatically parallelizes them, generates efficient MPI/C++ code, and provides resiliency. Since HPAT is compiler-based, it avoids overheads of library-based systems such as dynamic task scheduling and master-executor coordination. In addition, it provides automatic optimizations for scripting programs, such as fusion of array operations. Therefore, HPAT is $14\times$ to $400\times$ faster than Spark on the Cori supercomputer at LBL/NERSC. Furthermore, HPAT is much more flexible in distributed data structures, which enables the use of existing libraries such as HDF5, ScaLAPACK, and Intel® DAAL.

1. Introduction

Big data analytics applies advanced analytics and machine learning techniques to gain new insight from large data sets. These large data sets can be from various sources such as sensors, web, log files, and social media. Big data analytics allows users to take advantage of the available data to extract knowledge and make better and faster decisions. However, supporting fast decision making necessitates rapid development of the application by domain expert programmers (i.e., high productivity) and low execution time (i.e., high performance).

There are several productivity and performance considerations for an ideal big data analytics framework. For instance, high productivity in data analytics domain requires scripting languages such as MATLAB, R, Python, and Julia since they facilitate expressing mathematical operations and are the most productive languages in practice [1,2]. Furthermore, the framework should provide high performance on large-scale distributed-memory clusters due to extreme dataset sizes. Moreover, reliability is necessary since iterative machine learning algorithms can have long execution times.

Currently, there is a significant productivity and performance gap in the big data analytics domain. Existing big data analytics frameworks such as Apache Hadoop [3] and Apache Spark [4] enable productive big data analytics on clusters using the MapReduce programming paradigm [5]. MapReduce provides high-level parallelism abstractions suitable for data-parallel analytics programs, which can also be provided on top of scripting languages. However, this productivity comes at the cost of losing performance; these frameworks are orders of magnitude slower than hand-written MPI/C++ programs [6–8]. Another option is writing low-level MPI/C++ codes which is highly labor-intensive and not practical for data scientists. A fundamental issue is that these frameworks are library-based, requiring a runtime system to coordinate parallel execution across all the nodes by dynamic task scheduling. This leads to high runtime overheads - for example, the master node managing execution is typically a bottleneck.

To provide productivity and performance simultaneously, we propose automatic compilation of high-level scripting programs to efficient low-level parallel codes. However, such a challenging approach would require robust static analysis of high-level programs and effective parallelization and data partitioning without creating runtime overheads such as master-executor bottlenecks.

In this paper, we present High Performance Analytics Toolkit (HPAT)¹, which solves this problem using domain-specific compilation techniques. HPAT is a compiler-based framework for big data analytics on large-scale clusters that automatically parallelizes analytics tasks and gener-

¹HPAT is available online as open-source at <https://github.com/IntelLabs/HPAT.jl>.

ates scalable and efficient MPI/C++ code. In addition, HPAT offers flexibility in distributed data structures (even two-dimensional partitioning instead of one-dimensional when necessary) which enables using existing HPC libraries such as HDF5 [9], ScaLAPACK [10], and Intel® Data Analytics Acceleration Library (Intel® DAAL) [11]. Furthermore, HPAT provides resiliency using automatic checkpointing and facilitates optimization and fusion of array operations.

Automatic distributed-memory parallelization is recognized as a very difficult problem. To the best of our knowledge, HPAT is the first compiler-based system that can parallelize analytics programs automatically and achieve high performance. This is made possible by:

- A novel system design (Section 3)
- Domain-specific partitioning inference and parallelization (Section 4)
- Parallel I/O code generation (Section 3.1)
- Automatic checkpointing (Section 6)

Our evaluation demonstrates that HPAT is $14\times$ to $400\times$ faster than Spark on 64 nodes of the Cori supercomputer [12] and provides similar performance to hand-written MPI/C++ programs (Section 8). HPAT also scales better to larger number of nodes. We provide performance analysis for different benchmarks to gain insight about the performance differences.

2. Pi Example

We introduce HPAT and compare it to Spark using what is typically Spark’s first example program, a Monte Carlo π estimation program called *Pi* (Figure 1a). Spark (Python interface) is chosen as baseline due to its higher productivity and performance compared to other frameworks [13]. The *Pi* program generates a large number of random points on a two-dimensional unit square and counts the number of points inside its inscribed quarter circle. Then, the program uses the ratio of the areas of these two geometric shapes to estimate the value of π .

The Spark code for the *Pi* example is executed as follows. The *master* node starts the program. When the program reaches the `sc.parallelize()` call in line 16, the program initializes a resilient distributed dataset (RDD) using the provided range object. An RDD is linear distributed collection that is managed by Spark’s runtime system. However, the program does not instantiate the RDD data elements yet due to the *lazy evaluation* optimization. The next operation is a *map* where function `f()` of line 7 is called on every element of the RDD. The resulting RDD still does not need to be instantiated. The next operation is a *reduce* which causes the RDD to be instantiated since the result is returned to the user context. At this point, the task scheduler of *master* divides the tasks (of size *partition*) among the *executor* nodes, serializes the required task information such as *map* and *reduce* functions, and waits for the reduction to be completed.

```
1 from pyspark import SparkContext
2
3 if __name__ == "__main__":
4     sc = SparkContext(appName="PythonPi")
5     n = 100000 * partitions
6
7     def f(_):
8         x = random() * 2 - 1
9         y = random() * 2 - 1
10        return 1 if x ** 2 + y ** 2 < 1 else 0
11
12    def add(x, y):
13        x += y
14        return x
15
16    count = sc.parallelize(range(1, n + 1), partitions)
17               .map(f).reduce(add)
18    myPi = 4.0 * count / n
19    sc.stop()
```

(a) Spark Pi example

```
using HPAT

@acc hpat function calcPi(n)
    x = rand(n) * 2 - 1
    y = rand(n) * 2 - 1
    return 4.0*sum(x.^2 + y.^2 .< 1)/n
end

myPi = calcPi(10^9)
```

(b) HPAT Pi example

```
double calcPi(int64_t N) {
    int mpi_rank, mpi_nprocs;
    MPI_Comm_size(MPI_COMM_WORLD, &mpi_nprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &mpi_rank);
    int mystart = mpi_rank*(N/mpi_nprocs);
    int myend = (mpi_rank+1)*(N/mpi_nprocs);
    int64_t local_sum = 0;
    for(i=mystart; i<myend; i++) {
        x_tmp = rand() ...
        y_tmp = rand() ...
        local_sum += ...
    }
    MPI_Allreduce(...);
    return 4.0*global_sum/N;
}
```

(c) MPI/C++ code for Pi example (simplified demonstration)

Figure 1: Pi example

Figure 1b demonstrates the equivalent HPAT program, which uses standard Julia mathematical operations on vectors and does not include any explicit parallelism. The data-parallel task is written as a function decorated with `@acc hpat`. This macro instructs the compilation pipeline of HPAT to be applied to the function. HPAT uses domain-specific compilation techniques to extract the implicit parallelism and optimize the program. After compilation, an optimized MPI/C++ version of the function is generated and compiled into a binary by the system’s MPI compiler. The resulting program is similar to a scalable and efficient program written by an HPC programmer (demonstrated in Figure 1c). Note that semantically, the data-parallel Julia program creates 10 arrays. However, all of the data-parallel operations plus the

```

1 using HPAT
2
3 @acc hpat function logistic_regression(iters,
4     file_name)
5     points = DataSource(Matrix{Float64}, HDF5, "points",
6         file_name)
7     labels = DataSource(Vector{Float64}, HDF5, "labels",
8         file_name)
9     D, N = size(points)
10    labels = reshape(labels, 1, N)
11    w = reshape(2*rand(D)-1, 1, D)
12    for i in 1:iters
13        w -= ((1./(1+exp(-labels.*(w*points))))-1).*labels
14            *points'
15    end
16    return w
17 end
18
19 weights = logistic_regression(100, "mydata.hdf5")

```

Figure 2: HPAT logistic regression example.

reduction are fused into a single loop by HPAT, and the iteration space is divided among nodes in Single Program Multiple Data (SPMD) fashion. In a nutshell, HPAT is faster than Spark since it can optimize the program through compilation, and avoids runtime library overheads.

Our approach is generally effective due to the following characteristics of the data analytics domain. First, we observe that parallelism is often simple in this domain; the map/reduce parallelism pattern and one-dimensional data decomposition are predominantly assumed by data analytics frameworks. In addition, the computation and data structures are often well-structured and there is no dynamic behavior such as runtime load variation (except graph analytics). Second, high-level scripting programs are already implicitly parallel since parallelism and data communication can be inferred from array-style mathematical operations (Section 4).

HPAT is implemented as a Julia package and extends the Julia language (which is similar to MATLAB). HPAT programs take advantage of Julia’s high-level matrix and vector operations as well as HPAT-introduced domain-specific extensions for big data analytics. We chose Julia since its built-in type inference and support for introspection and metaprogramming make compiler construction easier, but the same techniques could be applied to other scripting languages such as MATLAB, R, and Python.

3. HPAT Overview

In this section, we provide an overview of the HPAT compilation pipeline. In the following sections, we discuss how the primary aspects of our system are implemented in this pipeline including automatic Parallel I/O (Section 3.1), distributed-memory translation (Section 3.2), using distributed-memory libraries (Section 5), and checkpointing (Section 6). We use the HPAT logistic regression example in Figure 2 and the K-Means example in Figure 3 to illustrate how HPAT parallelizes programs and generates MPI communication calls automatically. Note that HPAT programs are standard

```

1 using HPAT
2
3 @acc hpat function kmeans(numCenter, iters, file_name)
4     points = DataSource(Matrix{Float64}, HDF5, "points",
5         file_name)
6     D, N = size(points)
7     centroids = rand(D, numCenter)
8     for l in 1:iters
9         dist = [Float64[sqrt(sum((points[:,i]-centroids[j,
10             :]).^2)) for j in 1:numCenter] for i in 1:N]
11         labels = Int[indmin(dist[i]) for i in 1:N]
12         centroids = Float64[ sum(points[j, labels.==i])/
13             sum(labels.==i) for j in 1:D, i in 1:
14                 numCenter]
15     end
16    return centroids
17 end
18
19 centroids = kmeans(5, 100, "mydata.hdf5")

```

Figure 3: HPAT K-Means example.

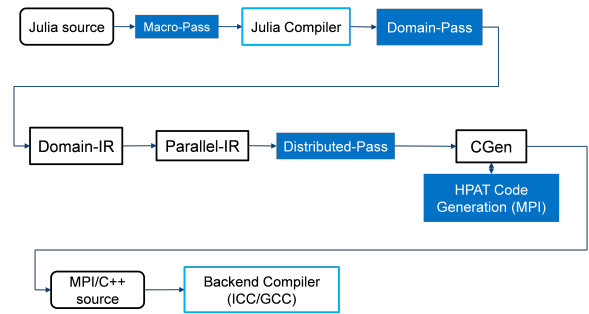


Figure 4: HPAT’s compilation pipeline.

Julia programs with a small number of extensions (e.g. *DataSource* syntax and *@acc hpat* annotation), and some limitations on coding style as described in Section 3.3.

HPAT uses the *@acc* macro provided by Julia’s CompilerTools package to configure HPAT’s compilation pipeline as shown in Figure 4. The solid boxes are HPAT components. The macro pass desugars HPAT extensions (such as *DataSource*) into function calls and type annotations to enable compilation by Julia. The Julia compiler then performs further desugaring and type inference on the function’s abstract syntax tree (AST). The Domain-Pass transforms HPAT extensions into a form more conducive to optimization by the subsequent Domain-IR and Parallel-IR passes. The Domain-IR and Parallel-IR passes are provided by Julia’s ParallelAccelerator package. The Domain-IR pass identifies operators (e.g., vector-vector element-wise addition) and other constructs in the AST whose semantics are parallelizable. The Parallel-IR pass takes the different kinds of parallel constructs found by Domain-IR and transforms those into a common representation called *parfor* (that represents a tightly nested *for* loop whose iterations can all be performed in parallel) and performs fusion and other AST optimizations. These two passes are described in more detail in Section 7. Given this parallel representa-

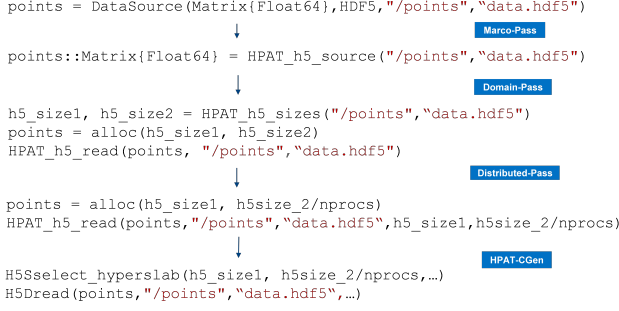


Figure 5: HPAT’s DataSource compilation pipeline.

tion, the Distributed-Pass partitions the arrays and *parfors* for distributed-memory execution and generates communication calls. HPAT Code Generation takes advantage of several hooks provided by *CGen* (part of ParallelAccelerator) to generate C++ code.

3.1 Automatic Parallel I/O

Figure 5 demonstrates how HPAT’s compilation pipeline translates *DataSource* syntax to parallel I/O code (*DataSink* is similar). Macro-Pass desugars the syntax into a HPAT placeholder special function call (e.g., *HPAT_h5_read*) and type-annotates the output array so that Julia’s type inference engine can infer types for the rest of the program. Domain-Pass generates function calls to get the size of the array and allocations so that Domain-IR and Parallel-IR can optimize the program effectively. Allocations and size variables are critical information that need to be explicit since many optimizations depend on them to be known. For example, operations on different arrays are fused into a single *parfor* and extra intermediate array allocations are removed only if arrays are known to have the same shape and size.

Distributed-Pass enables parallel I/O by partitioning the input array among nodes and adding the start and end indices for each dimension. HPAT Code Generation’s function call replacement mechanism generates the backend HDF5 code (currently MPI/C++) for placeholders such as *HPAT_h5_read*. HPAT also supports text files using MPI I/O because many big data files are stored as text. Note that parallel I/O codes in MPI/C++ (even libraries such as HDF5) are often hundreds of lines of low-level code and require understanding of many operations. Therefore, writing parallel I/O code manually is cumbersome and HPAT’s automation is significant.

3.2 Distributed-Memory Translation

The core compilation component of HPAT is the Distributed-Pass. The Distributed-Pass transforms the function for distributed-memory execution and generates communication calls. A key part of this pass is determining for each array or *parfor* whether it should be *sequential* (meaning each processor has a copy) or *partitioned* among processors (1D or 2D parti-

tioning). This pass starts by collecting information about the AST such as array shapes and sizes. The pass then performs a partitioning analysis for arrays and *parfors* as described in Section 4 to enable distributed-memory parallelization and using this information translates each AST node appropriately.

Other functions of this pass include the following. The pass inserts calls in the AST to query the number of processors and to get the node number, i.e. *hpat_dist_num_pes()* and *hpat_dist_node_id()*. Allocations of partitioned arrays are divided among nodes by inserting code to calculate size based on the number of processors (e.g. *mysize = total/num_pes*). *reshape()* operations are handled similar to allocations. Furthermore, size query calls for partitioned arrays (*arraysize()* and *arraylen()*) are translated to return the arrays global dimension sizes. Note that these size query calls are typically inserted in the AST in various compilation stages even if the user has not used them directly.

Parfors are translated by dividing the iterations among processors using node number and number of processors. Note that some *parfors* are not partitioned by our analysis and are therefore replicated on each processor. Array access indices in partitioned *parfors* are also updated accordingly. For example, $A[i]$ is replaced with $A[i - mystart]$ where *mystart* contains the starting index of the *parfor* on the current processor. Furthermore, for *parfors* with reductions, communication calls (e.g. *hpat_dist_allreduce()*) for distributed-memory reductions are generated.

The Distributed-Pass also translates “special” functions. For example, it fills out additional parameters to placeholder functions such as *HPAT_h5_read*, which requires start and end indices as additional inputs to facilitate later lowering to a selected backend. Furthermore, matrix/vector and matrix/matrix multiplication (GEMM calls) need special handling. For instance, $w * points$ in Figure 2 does not require communication since replication of w makes it data-parallel, while $(\dots * labels) * points'$ requires an *allreduce* operation since both arguments are partitioned. The Distributed-Pass makes this distinction by using the parallelization information provided by previous analyses. In the first case, the first input is sequential while the second input and the output are partitioned. In the second case, both inputs of GEMM are partitioned but the output is sequential. Section 4 contains more details about GEMM partitioning inference.

3.3 HPAT Coding Style

HPAT supports the high-level scripting syntax of the Julia language, which is intuitive to domain experts. However, users have to follow certain coding style guidelines to make sure HPAT can analyze and parallelize their programs automatically using heuristics:

- The analytics task should be written in functions that are annotated with `@acc hpat`.

- I/O (e.g. reading input samples) should be done through HPAT (using the *DataSource* and *DataSink* syntax).
- The data-parallel computations should be in the form of high-level matrix/vector computations or comprehensions since HPAT does not parallelize sequential loops.
- Julia’s column-major order should be followed for multidimensional arrays since HPAT parallelizes across last dimensions. For example, this means that features of a sample are in a column of the samples matrix.

4. Automatic Partitioning Analysis

The goal of HPAT’s partitioning analysis is to automatically parallelize common analytics tasks such as data-parallel queries and iterative machine learning algorithms using domain-specific heuristics. The main domain-specific heuristic is that partitioning of vectors and matrices is mainly one-dimensional (1D) for analytics tasks. This is how partitioning is performed in Spark for RDDs as well. In addition, map/reduce parallel semantics are assumed by default for mathematical operations. Furthermore, we assume by default that matrices are “tall-and-skinny” and are 1D partitioned across columns (since Julia is column-major). Note that these assumption do not apply to other domains. For example, multi-dimensional partitioning is often best in many physical simulation applications. Based on this foundation, we write rules for different operations based on their array access requirements. For example, data-parallel operations such as $A + B$ and $\log(A)$ (where A and B are arrays), which are translated to parfors, can have partitioned input and output if they are parallelized.

The automatic partitioning analysis algorithm of Distributed-Pass is demonstrated in Figure 6. For each array and parfor, Distributed-Pass infers whether it should be sequential (copied on different nodes) or partitioned among nodes in one-dimensional (1D) or two-dimensional (2D) manner. This algorithm is extensible to more partitioning paradigms as well. We define precedence among partitioning methods to ensure correct inference. Currently, sequential (SEQ) partitioning has highest precedence followed by 2D and 1D partitioning. 1D is lowest since it is assumed when starting inference. User involvement is required for 2D partitioning (described in section 4.1).

The algorithm repeatedly walks the abstract syntax tree (AST) until quiescence and for each node in the tree determines if the inference rules for that node causes an array or parfor to have a different partitioning (e.g. become sequential). Assignment AST nodes require both the left-hand side and right-hand side to have the same partitioning. Therefore, the partitioning with highest precedence is chosen (e.g. both are sequential if either one is). For return statements, the set of arrays being returned are each flagged as sequential since returned arrays need to fit on a single node and this output typically represents a summarization of a much larger data set. If larger output is required, the program should write the

```

Procedure partitioningAnalysis(state)
  initPartitioningsAs1D(state)
  setUserPartitionings(state)
  while hasChanged(state) do
    | updatePartitioning(state)
  end
  return

Procedure updatePartitioning(state)
  for each node  $\in$  AST do
    if isAssignment(node) then
      | matchPartitioning(lhs, rhs, state)
    else if isReturn(node) then
      | // returned arrays are sequential
      | state[getReturnedArrays(node)]  $\leftarrow$  SEQ
    else if isKnownCall(node) then
      | updateKnownCall(state,node)
    else if isUnknownCall(node) then
      | // assume all arrays are
      | sequential
      | state[nodeArrays]  $\leftarrow$  SEQ
    else if isGEMM(node) then
      | applyGemmRules(lhs,arr1,arr2,state)
    else if isParfor(node) then
      | applyParforRules(state, node)
    end
    | // ... handle other nodes
  end
  return

Procedure matchPartitioning(arr1,arr2,state)
  // assign partitioning with highest
  // precedence, e.g. arr1/arr2 are
  // sequential if either one is
  partitioning  $\leftarrow$ 
    maxPrecedence(state[arr1],state[arr2])
  state[arr1]  $\leftarrow$  state[arr2]  $\leftarrow$  partitioning
  return

```

Figure 6: Partitioning inference in HPAT.

```

Procedure applyGemmRules(lhs,arr1,arr2,state)
  if is1D(arr1,state)  $\wedge$  is1D(arr2,state)  $\wedge$ 
     $\neg$ isTransposed(arr1)  $\wedge$  isTransposed(arr2)
  then
    // e.g. ( $\dots$ *labels)*points' -
    // reduction across samples
    state[lhs]  $\leftarrow$  SEQ
  else if  $\neg$ is2D(arr1,state)  $\wedge$  is1D(arr2,state)  $\wedge$ 
     $\neg$ isTransposed(arr2)  $\wedge$  is1D(lhs,state) then
    // e.g. w*points - dot product with
    // sample features
    state[arr1]  $\leftarrow$  SEQ
  else if  $\neg$ isSEQ(arr1,state)  $\wedge$   $\neg$ isSEQ(arr2,state)
     $\wedge$   $\neg$ isSEQ(lhs,state)  $\wedge$  (is2D(arr1,state)  $\vee$ 
    is2D(arr2,state)  $\vee$  is2D(lhs,state)) then
    // If any array is 2D, all arrays
    // should be 2D
    state[lhs]  $\leftarrow$  state[arr2]  $\leftarrow$  state[arr1]  $\leftarrow$  2D
  else
    // all sequential if no rule applied
    state[lhs]  $\leftarrow$  state[arr2]  $\leftarrow$  state[arr1]  $\leftarrow$  SEQ
  return

```

Figure 7: HPAT inference rules for matrix multiply.

output to storage. This is a useful domain-specific heuristic that facilitates compiler analysis.

When the partitioning algorithm encounters a call site, the algorithm determines whether it knows the characteristics of the function being called. If so, the call site is considered to be a *known call*; otherwise, it is called an *unknown call*. For unknown calls, such as functions from external modules not compiled through HPAT, the partitioning algorithm conservatively assumes the involved arrays need to be sequential. If the function has parallel semantics for arrays, the user needs to provide the information. Conversely, partitioning inference rules are built into the algorithm for many Julia and HPAT operations. Common examples include Julia's array operations (e.g., reshape, array set, array get, array length), and HPAT's data storage functions.

Matrix/matrix and matrix/vector multiply operations (GEMM call nodes) are unique since they have more complex rules. Figure 7 demonstrates the rules using the logistic regression example of Figure 2. These formulas are derived based on our matrix partitioning and layout assumptions, and semantics of GEMM operations. Since samples are laid out in columns and we do not split sample features across processors (1D partitioning heuristic), any vector or row of matrix computed using reduction across samples is inferred as sequential. For example, the result of the inner formula in Logistic Regression's kernel is multiplied by the transpose of sample points and used to update the parameters ($w - = (\dots * labels) * points'$). Using the rule, Distributed-

Pass infers that the output of the operation is sequential if both inputs are partitioned and the second one is transposed. This also means that the inputs can stay partitioned. Note that in this case, a reduction is also inferred (which eventually turns into MPI_Allreduce in the backend). Furthermore, since the vector w is used in a dot product with matrix columns in $w * points$, it should be sequential and the matrix stays as 1D partitioned.

```

Procedure applyParforRules(parfor,state)
  partitioning  $\leftarrow$  1D
  myArrays  $\leftarrow$   $\emptyset$ 
  arrayAccesses  $\leftarrow$ 
    extractArrayAccesses(parfor.body)
  parforIndexVar  $\leftarrow$ 
    parfor.LoopNests[end].index_var
  for each arrayAccess  $\in$  arrayAccesses do
    if parforIndexVar =
      arrayAccess.index_expr[end] then
      // array is accessed in parallel
      // by this parfor
      myArrays  $\leftarrow$  myArrays  $\cup$ 
        arrayAccess.array
      partitioning  $\leftarrow$  maxPrecedence(par,
        state[arrayAccess.array])
    end
    if parforIndexVar  $\in$ 
      arrayAccess.index_expr[1:end-1] then
      // make parfor sequential if
      // parfor's last index variable
      // is used in accessing any lower
      // dimension of array
      partitioning  $\leftarrow$  SEQ
    end
    if isDependent(arrayAccess.index_expr[1:end],
      parforIndexVar) then
      // make parfor sequential if any
      // index is indirectly dependent
      // on parfor's index variable
      partitioning  $\leftarrow$  SEQ
    end
  end
  state[parfor]  $\leftarrow$  partitioning
  state[myArrays]  $\leftarrow$  partitioning
  return

```

Figure 8: HPAT inference rules for parfors.

As described in Section 7, *parfor* nodes represent the parallelism within a function and as such require special handling during partitioning inference. Figure 8 illustrates the partitioning inference function for *parfors*. As with array partitioning, we start by assuming that the *parfor* has 1D distribution until proven otherwise. Then, we analyze the body

of the *parfor* and extract all the array access/indexing operations (e.g., `some_array[index_expr1,...,index_exprN]`). We then iterate across all the discovered array accesses. Since HPAT parallelizes arrays and *parfors* across the last dimension, the index variable of the last loop of the *parfor* is used for testing. In the first case, we check if the index expression for the last dimension of the array access (i.e., `index_exprN`) is identical to the *parfor*'s index variable allocated to the last dimension of the loop nest. If so and the array being accessed is sequential then the *parfor* itself becomes sequential (partitioning with highest precedence is used). In the second case, we check whether the last dimension's *parfor* loop index variable is used directly or indirectly (e.g., `temp = parfor.LoopsNests[end].index_var; index_expr1 = temp + 1`) in any of the array access index expressions for the first $N-1$ dimensions. If so, then the *parfor* must be sequential. Note that these tests are conservative but do not typically prevent parallelization of common analytics codes we target with HPAT. If a program does not fit our heuristics, some programmer involvement is required. For example, it might be beneficial to distribute a *parfor* even with sequential arrays which may need expensive communication to keep arrays synchronized on every node.

Algorithm: SGD

```

samples ← read.input()
w ← initial.w()
for each iteration do
    | w ← gradient.update(samples, w)
end
return w

```

Figure 9: Typical stochastic gradient descent (SGD) method.

A main reason HPAT's heuristics are effective is that data analytics programs typically produce a summary of large data sets. In the case of machine learning algorithms, this summary is the weights of the trained model. More specifically, many large-scale machine learning algorithms use back-propagation and optimization methods such as stochastic gradient descent (SGD) [14, 15]. Hence, their structure can be represented as in Figure 9. Parameter set w is updated iteratively using back-propagation in order to minimize a form of energy function on samples. HPAT's analysis can infer that *samples* is partitioned since it will be accessed in data parallel manner. It will also infer that w is sequential since it is updated using a form of reduction. For example, variable w in Figure 2 is updated by a matrix/vector multiplication that implies a reduction. The reduction can be more indirect, however. Variable *centroids* of Figure 3 is a more complex example. Nevertheless, HPAT's analysis of array access patterns can infer the sequential access for this example as well.

```

1 using HPAT
2
3 @acc hpat function matrix_multiply(file1,file2,file3)
4     @partitioned(M,HPAT_2D);
5     M = DataSource(Matrix{Float64},HDF5,"/M", file1)
6     x = DataSource(Matrix{Float64},HDF5,"/x", file2)
7     y = M*x
8     y += 0.1*randn(size(y))
9     DataSink(y,HDF5,"/y", file3)
10 end

```

Figure 10: HPAT matrix multiply example.

4.1 Compiler Feedback and Control

A major drawback of many compiler-based approaches is that if some compiler analysis fails, the user program cannot be optimized effectively. This issue is even more critical for HPAT since HPAT's failure to parallelize and optimize effectively can lead to program failure since big data analytics programs can run out of memory easily. We address this issue by providing compiler feedback and explicit control to the user.

Partitioning and parallelization inference provides feedback about which arrays and operations are inferred as sequential and which inference rule (e.g. unknown call) caused it. Programmers can then change the inference behavior of HPAT by either changing their code (e.g. use higher-level abstractions) or providing hints to the compiler to infer particular arrays/*parfors* as sequential/partitioned. Furthermore, programmers can write explicitly parallel code using schemes such as *map/reduce* and *parallel for*.

User involvement is necessary for 2D partitioning cases since the inference algorithm cannot infer 2D partitioning easily. Consider the example program in Figure 10 (a real-world case requested by a user). The program reads two (multi-terabyte) matrices, multiplies them, adds some random value to the result, and writes it back to storage. The user specifies that matrix M requires 2D partitioning. HPAT infers that x and y matrices also need 2D partitioning as well. Furthermore, the related intermediate variables in the AST (such as the random matrix created) and the *parfors* are also 2D partitioned. In the backend, HPAT generates MPI/C++ code which calls parallel HDF5 for I/O and ScaLAPACK (PBLAS component) for matrix multiplication.

Manually developing efficient parallel code for this program is challenging. ScaLAPACK requires block-cyclic partitioning of input and output data but HDF5 provides a block-based read/write interface for parallel I/O. Hence, the generated code has to set 384 indices, since there are three I/O operations; each operation requires four hyperslab selections including corner cases, and each hyperslab selection requires start, stride, count and block size indices for two dimensions. In addition, ScaLAPACK's legacy interface is Fortran-based and has its own intricacies. As a result, the generated MPI/C++ code is 525 lines. Therefore, manually

```

using HPAT

@acc hpat function calcKmeans(k::Int64, file_name)
    points = DataSource(Matrix{Float64}, HDF5, "/points", file_name)
    clusters = HPAT.Kmeans(points, k)
    return clusters
end

```

Figure 11: HPAT library call example.

developing this code is highly error-prone for domain experts and HPAT’s automation is significant.

This use case demonstrates a fundamental advantage of our compiler approach. Library-based frameworks are based on fixed distributed data structures with specific partitioning and layout formats. For example, Spark’s RDDs are 1D partitioned and the whole framework (e.g block manager and scheduler) is based on this 1D partitioning. Supporting different partitionings and layouts is easy for a compiler, but is difficult for a library.

5. Automatic Utilization of Distributed-Memory Libraries

One of the most important use cases of analytics platforms is using libraries which include machine learning algorithms. They help the user gain insight from big data with minimal effort. For example, Spark provides MLlib [16] which is implemented using its RDD data format.

Since HPAT is compiler based, it can take advantage of distributed-memory libraries without requiring changes to their data structures and interfaces. On the other hand, only libraries that implement interoperability with Spark’s RDDs can be used with Spark. As section 4.1 demonstrated, this is a fundamental limitation for library-based frameworks since they have fixed distributed data structures. HPAT only needs MPI/C++ code generation routines to take advantage of libraries.

Figure 11 shows example code that calls a library. This function call goes through the compilation pipeline as a special known function call. Liveness analysis is aware that it does not change its inputs. In addition, partitioning analysis knows that the input array should typically be 1D partitioned, while the output array is sequential. If parallelization is successful, Distributed-Pass adds two new arguments to the function call; the first index and the last index of the input array on each node. HPAT’s CGen extension uses a MPI/C++ code routines for code generation of each call that is filled with the arguments provided. Currently, HPAT uses Intel® Data Analytics Acceleration Library (Intel® DAAL) [11] as an alternative to MLlib for machine learning algorithms.

HPAT’s automation is significant even for this simple program. Figure 12 demonstrates a small portion of the equivalent program in MPI/C++. The resulting program is hun-

```

...
kmeans::init::Distributed<step1Local, double, kmeans::
    init::randomDense> localInit(nClusters, nBlocks *
    nVectorsInBlock, rankId * nVectorsInBlock);
localInit.input.set(kmeans::init::data, dataSource.
    getNumericTable());
localInit.compute();
services::SharedPtr<byte> serializedData;
InputDataArchive dataArch;
localInit.getPartialResult()->serialize( dataArch );
size_t perNodeArchLength = dataArch.getArchiveSize(
    );
if (rankId == mpi_root)
{
    serializedData = services::SharedPtr<byte>( new byte[
        perNodeArchLength * nBlocks ] );
}
byte *nodeResults = new byte[ perNodeArchLength ];
dataArch.copyArchiveToArray( nodeResults,
    perNodeArchLength );
MPI_Gather( nodeResults, perNodeArchLength, MPI_CHAR,
    serializedData.get(), perNodeArchLength,
    MPI_CHAR, mpi_root, MPI_COMM_WORLD);
...

```

Figure 12: MPI/C++ equivalent to Figure 11 (a small sample).

dreds of lines which is highly cumbersome to write manually since it requires the programmer to understand the objects, function call requirements, and class hierarchy of the library.

6. Checkpointing

HPAT also provides automated application-level checkpointing and restart capabilities, targeted at iterative machine learning applications like logistic regression and k-means, which have a structure similar to Figure 9. For these applications, one only needs to checkpoint the learning parameters (w) being trained in the iteration loop and the loop index since the data points are read-only. Furthermore, since these learning parameters are replicated and synchronized on all processors, only one processor needs to checkpoint them. Therefore, HPAT checkpointing assumes a typical analytics function containing a single outer-loop and having the form: initialization, iteration loop, results. In the initialization phase, input is loaded and variables initialized, which establish the invariants for entry into the loop. The body of the outer-loop can be arbitrarily complex including containing nested loops. In the results phase, the outputs of the loop are used to compute the final result of the function.

To use automated checkpointing, the programmer annotates the function with `@acc hpat_checkpoint` rather than `@acc hpat`. HPAT then adds a checkpointing pass to the compilation pipeline after Domain-Pass. The checkpointing pass first locates the outer-loop and analyzes it to determine which variables are live at entry to the loop (including the loop index) and are written in the loop. This set of iteration-dependent variables are those saved as part of a checkpoint. The pass creates a new checkpoint function tailored to this set of variables and inserts a call to that function as the

first statement of the loop. In this function, MPI rank zero compares the time since the last checkpoint was taken with the next checkpoint time as calculated using Young’s formula [17]. If it is time to take a checkpoint, rank zero calls the HPAT runtime to start a checkpointing session, write each variable to the checkpoint, and then end the session. The HPAT runtime records the time to take the checkpoint and uses this information to improve the estimated checkpoint time that is input to Young’s formula. At the exit of the outer-loop, the pass also inserts a call to the HPAT runtime to indicate that the checkpointed region has completed and that any saved checkpoints can be deleted.

To restart a computation, the programmer calls the HPAT restart routine and passes the original function that terminated prematurely along with the original set of arguments to that function. When the HPAT compiler sees a call to restart, the compiler creates a specialized restart version of the function (that includes the checkpointing code) that is identical to the original but with the addition of checkpoint restore code added before the entry to the loop. This checkpoint restore code finds the saved checkpoint file and loads the iteration-dependent variables from the checkpoint. In this way, when a prematurely terminated function is restarted, the initialization code is performed again and the checkpoint restore then fast-forwards to the last successfully checkpointed iteration of the loop. By repeating the initialization code, variables used in the loop but not written by the loop (and therefore not stored in the checkpoint) are restored. As far as we are aware, this domain-specific approach of re-executing the initialization phase is novel although our approach is similar to Wang, et. al [18, 19] except that we store only live variables updated by the loop and not all live variables. Consider the logistic regression example in Figure 2; we store only the loop index i and w in the checkpoint whereas the full set of live data would include *points* and *labels* and would result in checkpoints orders of magnitude larger than our own.

In summary, HPAT provides resiliency for iterative machine learning applications. It uses compiler analysis to determine the minimal set of checkpointed data and restores other variables by re-executing the function’s initialization phase. The main insight is that while a compiler is able to perform the analysis for this minimal checkpointing, a library approach like Spark is unable to do so.

7. ParallelAccelerator Infrastructure

Before HPAT can distribute parallel work across a cluster, the parallel work within an application must first be discovered. This extraction of parallelism is done through Julia’s ParallelAccelerator package, which focuses on parallelism within dense Julia array operations. ParallelAccelerator consists of three main compiler passes, Domain-IR, Parallel-IR, and CGen. Domain-IR looks for operations and other constructs in Julia’s AST that have different kinds of parallel semantics and then replaces those operations with equivalent

Domain-IR nodes that encode those semantics. Some of the most common parallelism patterns in Domain-IR are *map*, *reduce*, *Cartesian map*, and *stencil*. For example, Domain-IR would identify unary vector operations (such as $-$, $!$, \log , \exp , \sin , and \cos) and binary, element-wise vector-vector or vector-scalar operations (such as $+$, $-$, $*$, $/$, $==$, $!=$, $<$, and $>$) as having *map* semantics. Likewise, Julia’s *sum()* and *prod()* functions would be identified by Domain-IR as having *reduce* semantics. Domain-IR identifies comprehensions within the Julia code as having *Cartesian map* semantics.

The Parallel-IR pass lowers Domain-IR nodes to a common representation called *parfor*. Once in this representation, Parallel-IR performs *parfor* fusion between *parfors* coming from potentially dissimilar Domain-IR nodes. This fusion process reduces loop overhead and eliminates many intermediate arrays, helping the program to have better locality. There are three main components of the *parfor* representation: loop nests, reductions, and body. Every *parfor* has a loop nest that represents a set of tightly nested for loops. It is typical for the number of such loops to match the number of dimensions of the array on which the *parfor* operates. The *parfor* reductions component is only present when the *parfor* involves a reduction and encodes the variable holding the reduction value along with its initial value and the function used to combine reduction elements. Finally, the body of the *parfor* contains code to compute the result of the *parfor* for a single point in the iteration space. After the Parallel-IR pass is finished, CGen converts the AST to OpenMP-annotated C code in the backend.

8. Evaluation

We compare the performance of Spark, HPAT and hand-written MPI/C++ programs on the Cori supercomputer at NERSC [12]. We have similar results on a local cluster (not presented here). Evaluation on cloud environments with HPC support such as Amazon AWS [20] is left for future work. Cori is a Cray XC40 supercomputer that includes advanced features for data-intensive applications such as large memory capacity and high I/O bandwidth. Each node has two sockets, each of which is a 16-core Intel Haswell processor (Intel Xeon E5-2698 v3 2.3GHz). The memory capacity of each node is 128GB.

The default Spark 1.5.1 installation on Cori is used which is tuned and supported. We also found the performance of Spark 1.6.0 to be similar for our benchmarks. Benchmark sizes are chosen so that they fit in the memory, even with excessive memory usage of Spark, to make sure Spark’s performance is not degraded by accessing disks repeatedly. HPAT is capable of generating MPI/OpenMP but currently, it turns OpenMP off and uses MPI-only configuration since OpenMP code generation is not tuned yet². We do not use hyperthreading and just run one MPI rank per core. Table 1

²HPAT turns OpenMP on automatically for libraries that are tuned for it (not evaluate here).

Benchmark	Description	Input size/iterations
1D Sum	Sum a large vector read from file	8.5 billion double precision elements
1D Sum Filter	Filter and sum a large vector read from file	8.5 billion double precision elements, 80% filtered
Monte Carlo Pi	Estimate Pi using Monte Carlo method	1 billion random elements
Logistic Regression	iterative Logistic Regression algorithm on data read from file	2 billion 10-feature single precision elements, 200 iterations
K-Means	K-Means clustering algorithm on data read from file	320 million 20-feature double precision elements, 10 iterations, 5 centers

Table 1: Description of benchmarks.

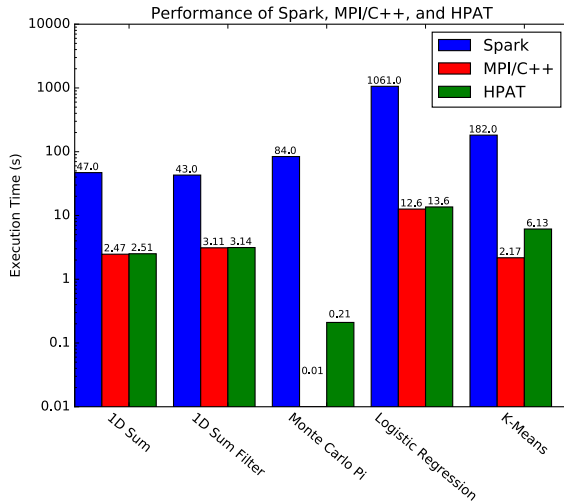


Figure 13: Performance comparison for benchmarks on 64 nodes.

describes the benchmarks and their input sizes. Note that we only compare the performance of manually written codes and not machine learning libraries. The Spark versions of the benchmarks are mostly available in Spark’s open-source distribution. The MPI/C++ programs (which the authors developed) simply divide the problem domain across ranks equally, and ensure maximum locality by fusing the loops manually. Parallel I/O times are included in all results, but they are not significantly different across systems. MPI/C++ codes are about $6\times$ longer in lines of code compared to HPAT codes.

Figure 13 compares the performance of Spark, manual MPI/C++, and HPAT on 64 nodes (2048 cores). HPAT is $14\times$ - $400\times$ faster than Spark for the benchmarks and provides similar performance to MPI/C++. The $14\times$ speedup for *1D Sum Filter* is important since filtering data and gathering statistics is very common in data analytics.

Monte Carlo Pi is the most extreme; HPAT is $400\times$ faster than Spark. The reason is that computation per element is small and the Spark overheads such as task scheduling and master-executor coordination are amplified. More importantly, Spark creates a large RDD which holds the results of random points before performing the reduction (see Figure 1a), while HPAT eliminates the unnecessary array. This results in significant locality difference (memory compared

to registers). In principle, Spark could fuse the map and reduce operations and not create the RDD. However, a library cannot decide to perform this optimization easily since it has no way of knowing whether the RDD will be reused later in the program. This is an example of fundamental limitations of lazy evaluation that a compiler avoids. Furthermore, HPAT is slower than the manual MPI/C++ code even though the generated code is structurally equivalent. The issue is that the generated loop is much longer since it includes many extra intermediate (scalar) variables. Therefore, the backend C++ compiler cannot optimize and vectorize it effectively. We plan to resolve this issue by a copy propagation pass right before backend code generation (CGen).

Logistic Regression is $78\times$ faster in HPAT than Spark. One significant overhead component for this benchmark is Python since it includes many Numpy matrix/vector computations in its kernel, which have less locality than fused C++ loops. To evaluate this difference, we compared Python and C++ kernels of Logistic Regression on a single core and found that Python is $6.5\times$ slower. In principle, Spark could integrate with a loop-fusing compiler and avoid this issue, but interfacing a compiler increases the complexity of the system. Furthermore, overhead of compilation during runtime can be significant. Our approach naturally avoids these issues. Note that the rest of performance difference is due to Spark’s runtime which is still more significant. HPAT is slightly slower than the MPI/C++ code for *Logistic Regression* since the matrix-vector products in the formula (see Figure 2) are expanded and fused with other loops in the manual code, while HPAT does not perform this optimization yet. This also increases the memory consumption of HPAT due to storing intermediate results.

Even though HPAT provides $30\times$ speedup for *k-means* compared to Spark, the generated code is $2.8\times$ slower than the hand-tuned MPI/C++ version. One issue is that our infrastructure is currently unable to fuse the loops computing *labels* and *centroids* variables since loop interchange is required. Furthermore, the hand-written code stores the local results for *centroids* in temporary variables and calls *MPI_Allreduce* out of the inner loops while HPAT generates *MPI_Allreduce* for each element of *centroids* separately due to the use of a comprehension for updating the elements. This increases the cost of communication initiation significantly. We plan to develop a communication optimization pass after Distributed-Pass that handles these cases by performing a set of communication-related transforma-

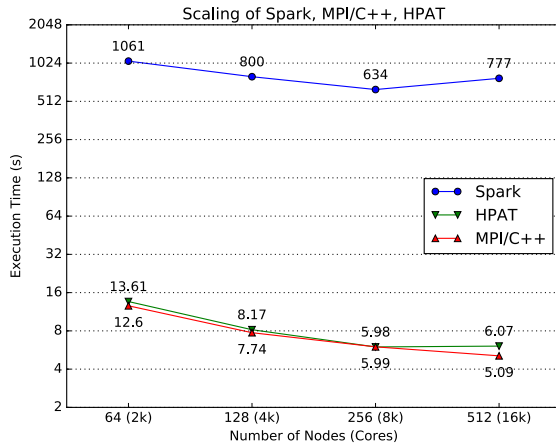


Figure 14: Strong scaling of Spark and HPAT for Logistic Regression.

tions such as hoisting communication calls out of the loops when possible.

Figure 14 demonstrates the strong scaling of Spark, MPI/C++ and HPAT for *Logistic Regression*. HPAT scales better than Spark and therefore, the performance gap is increased from 78x to 128x. The master-executor architecture of Spark causes several issues for strong scaling. First, the master is a bottleneck since it needs to divide the data in blocks and coordinate the tasks on all the executors (up to 16k cores in Figure 14). On the other hand, SPMD programs do not need a master and each process (MPI rank) implicitly knows which portion of the data and computation belongs to it. In addition, the results of the reduction (model weight updates) needs to go to the master before being broadcast to executors while *MPI_Allreduce* can decentralize this communication step. Moreover, we found that Spark divides the input data into 6313 blocks and the same number of associated tasks for this benchmark, irrespective of the number of processors. The underlying heuristic most probably intends to minimize the relative cost of data and task management compared to the actual computation. However, this causes large-scale configurations with more cores than 6313 to be inefficient due to lack of parallelism. We found that forcing Spark to create more tasks can increase runtime overheads substantially. More in depth analysis of Spark’s performance is out of this paper’s scope.

One notable advantage of our compiler approach is that it can decouple the programming from the execution environment. For example, HPAT users write Julia programs but the generated MPI/C++ code is almost universally supported on clusters. Conversely, Julia (and Python/Java for Spark) are less well supported in cluster environments. One disadvantage of our compiler approach is that it ties HPAT to single language whereas Spark’s library approach allows it to service multiple languages.

9. Related Work

Cai et al. evaluated the productivity of existing large-scale analytics frameworks and found Apache Spark (Python interface) to be the best in terms of productivity and performance [13]. However, previous studies have demonstrated that popular analytics frameworks such as Spark are orders of magnitude slower than hand-tuned MPI implementations [6–8], which provided motivation for our system. Several performance analysis studies for data analytics frameworks can be found in the literature [21–24]. To improve the performance of data analytics frameworks, some previous studies have focused on more efficient inter-node communication [25–27]. For example, Harp provides a MPI collective communication plugin for Hadoop [27]. However, they do not address the fundamental performance bottlenecks resulting from library implementations.

Other systems such as Pig [28] provide even higher-level programmer abstractions often at the cost of less performance. Efforts such as Pig-on-Spark [29] (i.e., Spork) are similar to ours in that they seek to maintain programmer productivity while automatically mapping to a higher-performant substrate. However, they do not compile and optimize the programs. Impala compiles SQL queries but cannot handle other programs [30]. TUPLEWARE compiles and optimizes user-defined function (UDF) workflows with limited top-level operators, but cannot handle general scripting programs [31].

Distributed Multiloop Language (DMLL) provides compiler transformations on MapReduce programs on distributed heterogeneous architectures [6]. However, it does not provide scripting abstractions and is not capable of generating efficient MPI/C++ code. Distributed Halide translates high-level stencil pipelines of image processing into parallel code for distributed-memory architectures [32]. However, the compiler techniques for image processing domain are different than the data analytics domain.

10. Conclusion and Future Work

Library-based big data analytics frameworks such as Spark provide programmer productivity but they are much slower than hand-tuned MPI/C++ codes due to issues such as high runtime overheads. We introduced an alternative compiler-based solution called High Performance Analytics Toolkit (HPAT). HPAT provides the best of both worlds: productivity of scripting abstractions and performance of efficient MPI/C++ codes. In addition, we introduced novel domain-specific compiler heuristics for data analytics domain that make our compiler approach possible. Our evaluation demonstrated that HPAT is 14x–400x faster than Spark. We plan to expand HPAT to provide more data analytics features and use cases. For example, providing support for sparse computations, data frames (heterogeneous tables), out-of-core execution is under investigation. Most of these

features need research on multiple layers; from scripting abstractions to compilation techniques and code generation.

References

- [1] L. Prechelt, “An empirical comparison of seven programming languages,” *IEEE Computer*, vol. 33, no. 10, pp. 23–29, Oct. 2000.
- [2] J. C. Chaves, J. Nehrbass, B. Guilfoos, J. Gardiner, S. Ahalt, A. Krishnamurthy, J. Unpingco, A. Chalker, A. Warnock, and S. Samsi, “Octave and python: High-level scripting languages productivity and performance evaluation,” in *HPCMP Users Group Conference, 2006*, 2006, pp. 429–434.
- [3] T. White, *Hadoop: The definitive guide*. O’Reilly Media, Inc., 2012.
- [4] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, “Spark: Cluster computing with working sets,” *HotCloud*, vol. 10, pp. 10–10, 2010.
- [5] J. Dean and S. Ghemawat, “Mapreduce: simplified data processing on large clusters,” *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.
- [6] K. J. Brown, H. Lee, T. Rompf, A. K. Sujeeth, C. De Sa, C. Aberger, and K. Olukotun, “Have abstraction and eat performance, too: Optimized heterogeneous computing with parallel patterns,” in *Proceedings of the 2016 International Symposium on Code Generation and Optimization (CGO)*, 2016, pp. 194–205.
- [7] J. L. Reyes-Ortiz, L. Oneto, and D. Anguita, “Big data analytics in the cloud: Spark on hadoop vs mpi/openmp on beowulf,” in *INNS Conference on Big Data 2015, San Francisco, CA, USA, 8-10 August 2015*, 2015, pp. 121–130.
- [8] F. McSherry, M. Isard, and D. G. Murray, “Scalability! but at what cost?” in *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*, 2015.
- [9] “The HDF5 file format and library,” <https://www.hdfgroup.org/HDF5/>.
- [10] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK Users’ guide*. Siam, 1999, vol. 9.
- [11] “Intel Data Analytics Acceleration Library,” <https://software.intel.com/en-us/daal>.
- [12] “Cori Supercomputer at NERSC,” <http://www.nersc.gov/users/computational-systems/cori/>.
- [13] Z. Cai, Z. J. Gao, S. Luo, L. L. Perez, Z. Vagena, and C. Jermaine, “A comparison of platforms for implementing and running very large scale machine learning algorithms,” in *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD ’14, 2014.
- [14] L. Bottou, “Large-scale machine learning with stochastic gradient descent,” in *Proceedings of COMPSTAT’2010*. Springer, 2010, pp. 177–186.
- [15] C. M. De Sa, C. Zhang, K. Olukotun, and C. Ré, “Taming the wild: A unified analysis of hogwild-style algorithms,” in *Advances in Neural Information Processing Systems*, 2015, pp. 2656–2664.
- [16] “Spark Machine Learning Library (MLlib) Guide,” <http://spark.apache.org/docs/latest/mllib-guide.html>.
- [17] J. W. Young, “A first order approximation to the optimum checkpoint interval,” *Commun. ACM*, vol. 17, no. 9, pp. 530–531, Sep. 1974.
- [18] P. Wang, X. Yang, H. Fu, Y. Du, Z. Wang, and J. Jia, “Automated application-level checkpointing based on live-variable analysis in mpi programs,” in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, ser. PPoPP ’08, 2008, pp. 273–274.
- [19] P. Wang, Y. Du, H. Fu, X. Yang, and H. Zhou, “Static analysis for application-level checkpointing of mpi programs,” in *High Performance Computing and Communications, 2008. HPCC ’08. 10th IEEE International Conference on*, Sept 2008, pp. 548–555.
- [20] “HPC Support on Amazon AWS Cloud,” <https://aws.amazon.com/hpc/>.
- [21] A. Javed Awan, M. Brorsson, V. Vlassov, and E. Ayguade, “Performance characterization of in-memory data analytics on a modern cloud server,” in *2015 IEEE Fifth International Conference on Big Data and Cloud Computing (BDCloud)*. IEEE, 2015, pp. 1–8.
- [22] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, and B.-G. Chun, “Making sense of performance in data analytics frameworks,” in *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, Oakland, CA, May 2015.
- [23] Z. Jia, J. Zhan, L. Wang, R. Han, S. A. McKee, Q. Yang, C. Luo, and J. Li, “Characterizing and subsetting big data workloads,” in *Workload Characterization (IISWC), 2014 IEEE International Symposium on*, 2014.
- [24] D. Loghin, B. M. Tudor, H. Zhang, B. C. Ooi, and Y. M. Teo, “A performance study of big data on small nodes,” *Proc. VLDB Endow.*, vol. 8, no. 7, Feb. 2015.
- [25] X. Lu, W. Rahman, N. Islam, D. Shankar, and D. Panda, “Accelerating spark with rdma for big data processing: Early experiences,” in *International Symposium on High Performance Interconnects (HotI’14)*, August 2014.
- [26] N. Islam, W. Rahman, X. Lu, D. Shankar, and D. Panda, “Performance characterization and acceleration of in-memory file systems for hadoop and spark applications on hpc clusters,” in *2015 IEEE International Conference on Big Data*, October 2015.
- [27] B. Zhang, Y. Ruan, and J. Qiu, “Harp: Collective communication on hadoop,” in *Cloud Engineering (IC2E), 2015 IEEE International Conference on*, 2015, pp. 228–233.
- [28] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins, “Pig latin: A not-so-foreign language for data processing,” in *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD ’08, 2008, pp. 1099–1110.
- [29] “Pig on Spark,” <https://cwiki.apache.org/confluence/display/PIG/Pig+on+Spark>.
- [30] M. Kornacker, A. Behm, V. Bittorf, T. Bobrovitsky, C. Ching,

- A. Choi, J. Erickson, M. Grund, D. Hecht, M. Jacobs *et al.*, “Impala: A modern, open-source sql engine for hadoop.” in *CIDR*, 2015.
- [31] A. Crotty, A. Galakatos, K. Dursun, T. Kraska, C. Binnig, U. Cetintemel, and S. Zdonik, “An architecture for compiling udf-centric workflows,” *Proc. VLDB Endow.*, vol. 8, no. 12, Aug. 2015.
- [32] T. Denniston, S. Kamil, and S. Amarasinghe, “Distributed halide,” in *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 2016, p. 5.